TÉCNICO LISBOA

Deep Learning

2024/2025 – 1st Semester, P2

1st Homework Report

Authors:

Francisco Carmo (103618) francisco.m.carmo@tecnico.ulisboa.pt
Francisco Tavares (103402) francisco.carreira.tavares@tecnico.ulisboa.pt

Contents

Group Contributions: In this project, we divided the work as follows: Francisco Tavares completed the first exercise, while Francisco Carmo was responsible for the second exercise. For the third exercise, we split the work evenly: Francisco Carmo worked on parts 3.1 and 3.2, while Francisco Tavares completed parts 3.3 and 3.4. Additionally, we reviewed the entire project together to ensure the accuracy and clarity of our work, and we took the time to understand all exercises, including those initially assigned to the other group member.

Question 1

1.1 - Perceptron Algorithm

To test the Perceptron algorithm, we implemented the `update_weight` function. This function adjusts the model's weights based on whether the predicted class matches the true label for a given training example. If the prediction is incorrect, the weights for the true class are updated by adding the feature vector scaled by the learning rate, while the weights for the incorrectly predicted class are penalized by subtracting the same value. This ensures that the decision boundary gradually shifts to correctly classify the example. As recommended, we trained the model for 100 epochs and monitored the training and validation accuracies throughout the process. As shown in Figure 1, the plot of training and validation accuracies over the epochs demonstrates a noticeable improvement in the training accuracy. However, the validation accuracy remained relatively consistent over time, which suggests a lack of model capacity to generalize further. The final validation accuracy obtained was 0.3743.

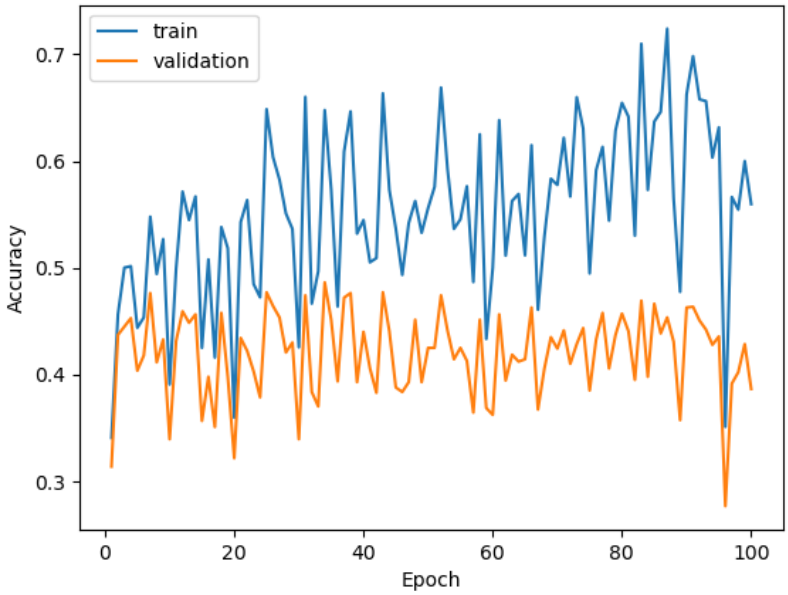


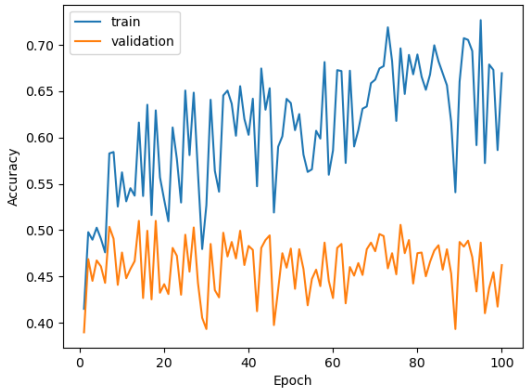
Figure 1: Training and Validation Loss over 100 epochs

1.2 - Logistic Regression with SGD

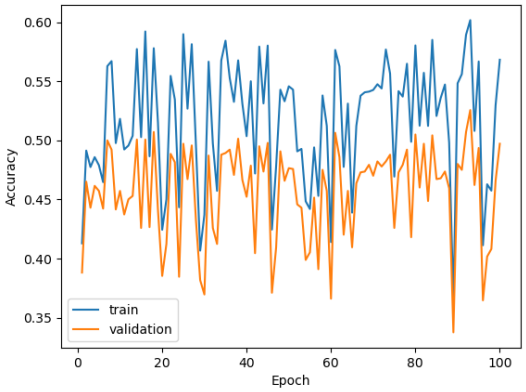
a) & b) - Losgistic Regression with and without Regularization

In this problem, we implemented Logistic Regression using stochastic gradient descent as the training algorithm. Initially, we developed the `update_weight` function to train a logistic regression model without regularization. Later, we modified the function to include L2 regularization, which penalizes large weights by adding a term proportional to their squared magnitude to the loss function, reducing overfitting and improving generalization.

The results, shown in Figure 2a and Figure 2b, highlight the impact of regularization. Without L2 regularization, the model achieved a training accuracy of 0.6694 and a validation accuracy of 0.4597, indicating overfitting as the model performed well on the training data but poorly on unseen data. With an L2 penalty of 0.01, the validation accuracy improved to 0.5053, while the training accuracy decreased to 0.5683. This demonstrates the trade-off of regularization, where some training performance is sacrificed to enhance the model's ability to generalize to new data.

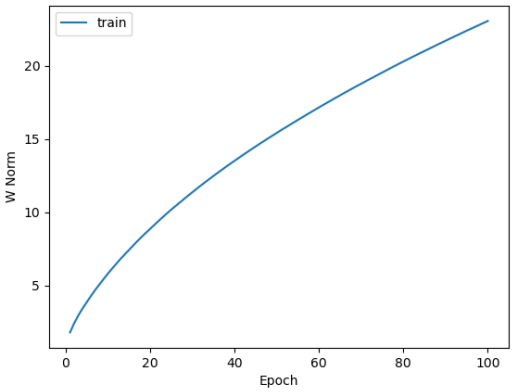


(a) Training and Validation Loss over 100 epochs for model without regularization

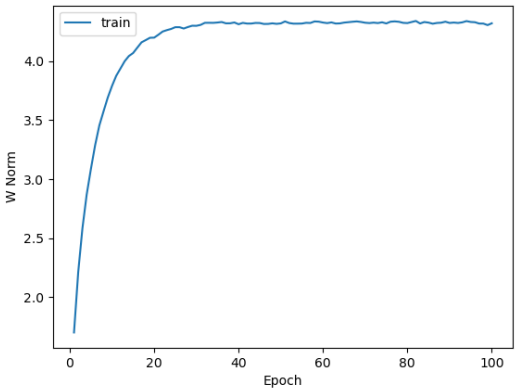


(b) Training and Validation Loss over 100 epochs for model with L2 regularization

b) - The Impact of L2 Regularization on Weight Magnitudes in Logistic Regression



(a) W norm over the epochs for the Logistics Regression without Regularization



(b) W norm over the epochs for the Logistics Regression Regularization

The L2 norm of the weights increased steadily during training for the non-regularized logistic regression model, reaching a relatively high value by the final epoch, as shown in Figure 3a. This reflects the model's tendency to focus heavily on a subset of features in order to minimize the training loss, leading to larger weights overall. Without regularization, the model overfits the training data, as large weights are used to achieve high training accuracy, like we discussed before at the expense of generalization.

In contrast, the L2 norm of the weights for the regularized model remained consistently lower throughout training due to the penalty term, as seen in Figure 3b. This constraint effectively limited the magnitude of the weights, ensuring they did not grow excessively large. Regularization helped improve the model's generalization ability, as evidenced by the increased validation accuracy.

The results emphasize the role of L2 regularization in controlling the complexity of the logistic regression model. By keeping the weights smaller, the regularized model achieved better generalization, albeit with slightly reduced training performance. This trade-off is a core feature of L2 regularization, which prevents overfitting and enhances robustness to unseen data.

c) - Expected Effects of L1 vs. L2 Regularization on Logistic Regression Weights

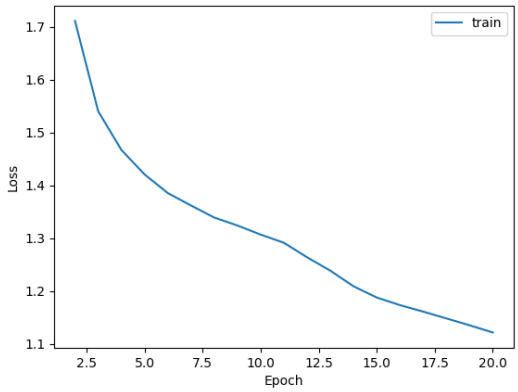
If L_1 regularization were used instead of L_2, the key difference would be the sparsity of the resulting weight vector. L_1 regularization tends to drive some weights to exactly zero, effectively performing feature selection by ignoring less important features, whereas L_2 regularization shrinks all weights uniformly without eliminating any completely. This sparsity improves interpretability and can enhance model efficiency, particularly in high-dimensional settings. However, the remaining non-zero weights in L_1-regularized models might be larger in magnitude compared to those in L_2-regularized models. Overall, L_1 regularization prioritizes a subset of key features, potentially improving generalization in scenarios where feature selection is critical, while L_2 provides a more uniform shrinkage that avoids complete elimination of weights.

1.3 - Multi Layer Perceptron

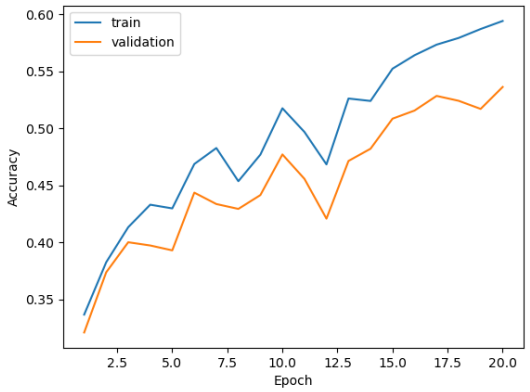
To solve this task, we modified the provided code skeleton as specified. The weights in the network were initialized using a normal distribution $w_{ij} \sim \mathcal{N}(\mu = 0.1, \sigma^2 = 0.1^2)$, and biases were initialized to zero. Training was performed using stochastic gradient descent (SGD) for 20 epochs with a learning rate of 0.001. Backpropagation was implemented to compute gradients, enabling weight and bias updates during each step of training.

By implicitly regularizing the model's weights through SGD and utilizing the ReLU activation function, the MLP effectively minimized training loss over epochs. The final validation accuracy achieved was **53.47%**, showcasing the model's moderate generalization capability on unseen data.

The results are summarized in the figures below. Figure 7a illustrates the reduction in training loss over the 20 epochs, while Figure 7b shows the corresponding increase in training and validation accuracies.



(a) Training loss evolution as a function of epoch number.

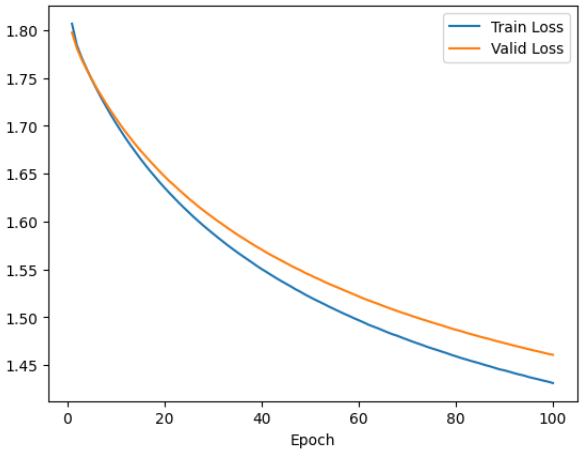


(b) Training and validation accuracies as a function of epoch number.

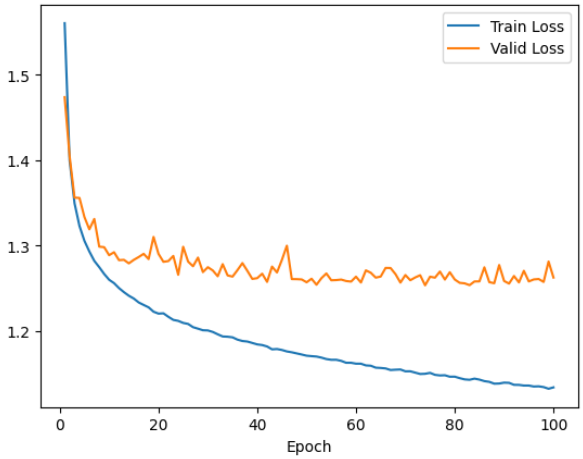
Question 2

2.1 - Logistic Regression

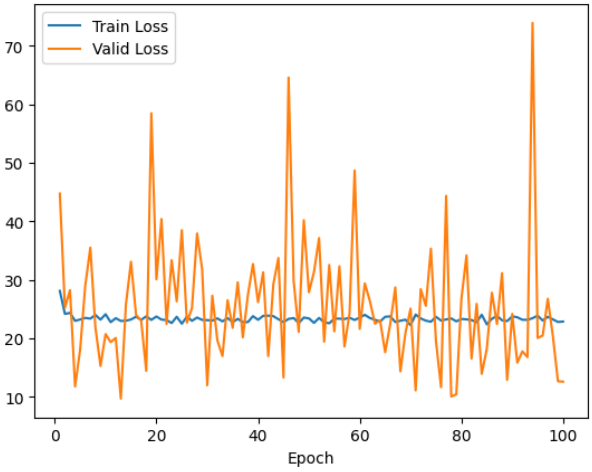
To evaluate the performance of a linear logistic regression model, we implemented the training process using stochastic gradient descent (SGD) with L_2 regularization ($l2\_decay = 0.01$) and a batch size of 32. The model parameters, including weights and biases, were optimized over 100 epochs, while tuning the learning rate across the following values: $\{0.00001, 0.001, 0.1\}$. The learning rate 0.001 demonstrated the best performance, achieving a validation accuracy of 0.5235 and a final test accuracy of 0.5250 (The learning rate 0.1 achieved a validation accuracy of 0.3426 and a final test accuracy of 0.3327 and the learning rate 0.00001 achieved a validation accuracy of 0.4480 and a final test accuracy of 0.4590). While $lr = 0.001$ achieved the best results within the 100-epoch limit, the $lr = 0.00001$ plot analysis revealed that a smaller learning rate might require more epochs to converge adequately because of a smaller difference between the validation and test losses which indicates less overfitting (the plot with the $lr = 0.1$ is a great example of overfitting and the results show that) . However, after additional tests with more epochs, which lowers the computational efficiency, the results for $lr = 0.00001$ still did not surpass the performance achieved with $lr = 0.001$, confirming the latter as the optimal choice for this task.



(a) Training and Validation Loss lr=0.00001



(b) Training and Validation Loss lr=0.001

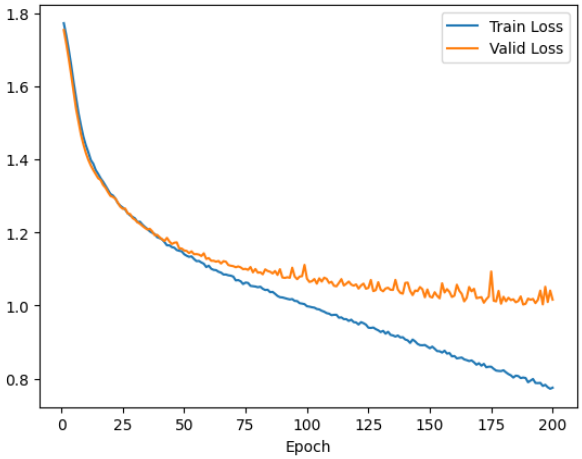


(c) Training and Validation Loss lr=0.1

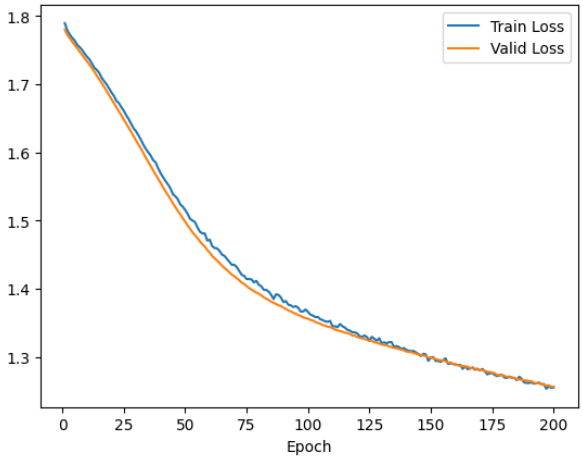
2.2 - Feed Forward Neural Network

a) - Impact of Batch Size on Model Performance and Execution Time

To evaluate the performance of two Feedforward Network models with different batch sizes, we trained one model using the default hyperparameters (batch size = 64) and another with a larger batch size (batch size = 512), keeping all other hyperparameters unchanged. The model with batch size = 64 achieved a validation accuracy of 0.6047 and a test accuracy of 0.6083 in 8 minutes and 20 seconds, outperforming the larger batch size (validation accuracy = 0.5028, test accuracy = 0.5190), which trained faster at 6 minutes and 16 seconds. Despite the improved efficiency, the larger batch had worst test and validation accuracies which translates to higher loss values in the plot (although it has less signs of overfitting so it should improve with more epochs).



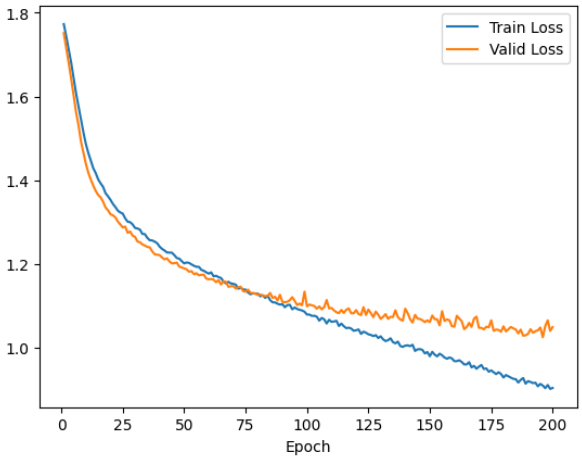
(a) Batchsize = 64



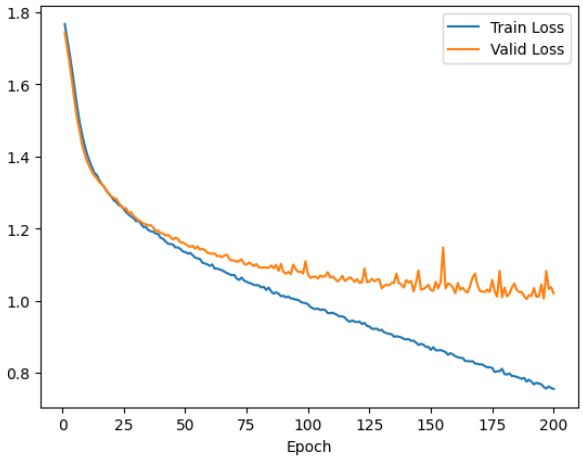
(b) Batchsize = 512

b) - Effect of Dropout Rates on Training and Validation Performance

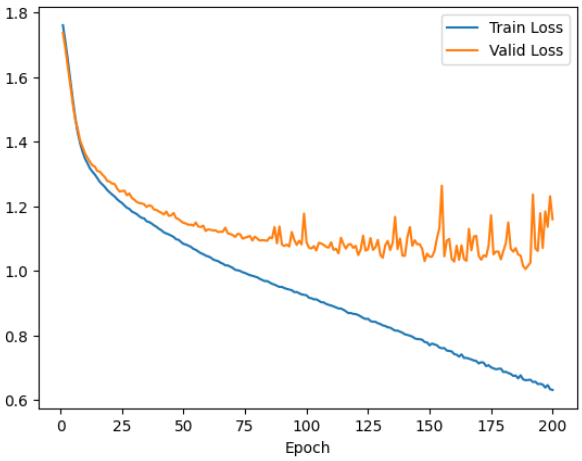
To evaluate the impact of different dropout rates on the performance of the model, we trained the model with three dropout values: 0.01, 0.25, and 0.5, keeping all other hyperparameters unchanged. The model with a dropout rate of 0.25 achieved the best performance, with a validation accuracy of 0.6047 and a test accuracy of 0.6060. This indicates that a moderate level of dropout helps the model generalize better and prevents overfitting. The configuration with a dropout rate of 0.01 showed the lowest performance (validation accuracy = 0.5734, test accuracy = 0.5747). This suggests that minimal regularization is insufficient to combat overfitting, leading to suboptimal generalization on unseen data. Increasing the dropout rate to 0.5 reduced test accuracy to 0.5927, slightly below the performance of the 0.25 configuration. Although this configuration demonstrates effective regularization, the higher dropout rate may be limiting its ability to learn and the increased dropout doesn't necessarily translate to better validation and test losses as we can see by the plots. The 0.01 dropout value was clearly the worst in terms of both having higher losses and lower accuracies.



(a) Dropout=0.5



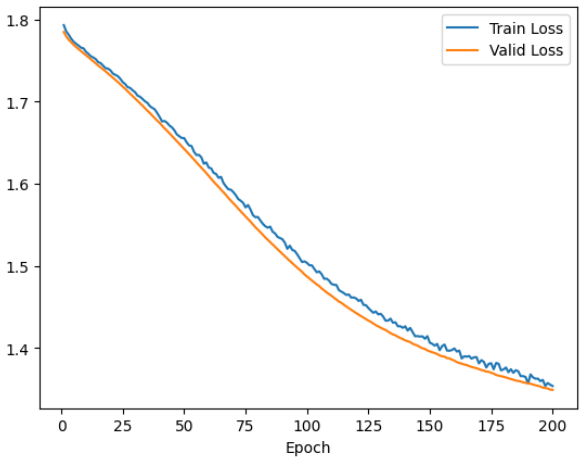
(b) Dropout=0.25



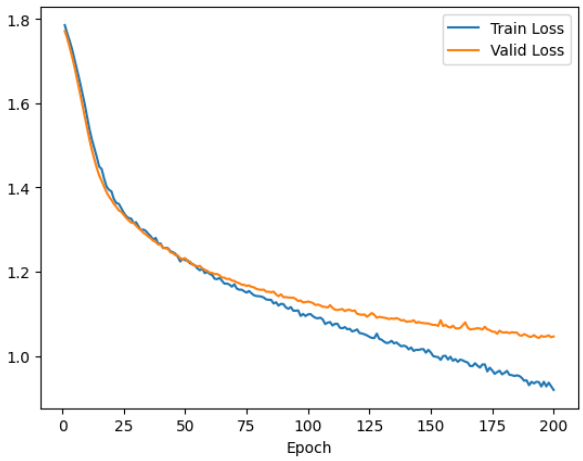
(c) Dropout=0.01

c) - Influence of Momentum on Training Dynamics and Accuracy

To evaluate the impact of the momentum parameter on the performance of a Feedforward Network, the default model was trained with a batch size of 1024 while setting the momentum to 0.0 and 0.9. Setting momentum to 0.9 significantly improved both validation (0.5983) and test 0.6010) accuracies compared to 0.0 (0.4701 and 0.4887, respectively). Without momentum, the optimizer relied solely on the current gradient, leading to slower convergence and possibly being stuck in local minima. Not only did we achieve better results with 0.9 momentum but we also needed less epochs to achieve lower losses and better both test and validation results.



(a) Momentum=0.0



(b) Momentum=0.9

Question 3

3.1 - Feature Transformation and Linear Representation of h

Knowing that h equals the activation function $g(z)$ defined as:

$$g(z) = z(1-z) = z - z^2,$$

where z is given by:

$$z = W_k x + b_k.$$

Substituting z into the expression for $g(z)$, we get:

$$g(z) = (W_k x + b_k) - (W_k x + b_k)^2.$$

Expanding the squared term, we obtain the following:

$$g(z) = W_k x + b_k - \left[(W_k x)^2 + 2b_k(W_k x) + b_k^2\right].$$

Rearranging the terms, the final expression for $g(z)$ is:

$$g(z) = b_k - b_k^2 + \sum_{j=1}^{D} \left(W_{kj} - 2b_k W_{kj}\right) x_j - \sum_{j=1}^{D}\sum_{l=1}^{D} W_{kj} W_{kl} x_j x_l.$$

The feature transformation $\phi(x)$ consists of the constant term 1 (representing the biases), the linear terms $x_1, \ldots, x_D$, and the quadratic terms $x_1^2, x_1x_2, \ldots, x_{D-1}x_D, x_D^2$. The constant term has dimension 1, the linear terms have D dimensions, and the quadratic terms have $\binom{D}{2} + D = \frac{D(D-1)+2D}{2} = \frac{D(D+1)}{2}$ dimensions.

Thus, the dimension of $\phi(x)$ is:

$$\dim(\phi(x)) = 1 + D + \frac{D(D+1)}{2} = \frac{(D+1)(D+2)}{2}.$$

The dimension of A_Θ is then:

$$\dim(A_\Theta) = K \times \dim(\phi(x)) = K \times \frac{(D+1)(D+2)}{2}.$$

From $g(z)$ we can define $\phi(x)$ as:

$$\phi(x) = \left[1, x_1, \ldots, x_D, x_1^2, x_1x_2, \ldots, x_D^2\right]^T.$$

and the matrix A_Θ that has the following structure:

$$A_\Theta = \begin{bmatrix} b_1 - b_1^2 & w_{11} - 2b_1w_{11} & \ldots & w_{1D} - 2b_1w_{1D} & -w_{11}^2 & -w_{11}w_{12} & \ldots & -w_{1D}^2 \\ b_2 - b_2^2 & w_{21} - 2b_2w_{21} & \ldots & w_{2D} - 2b_2w_{2D} & -w_{21}^2 & -w_{21}w_{22} & \ldots & -w_{2D}^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \\ b_k - b_k^2 & w_{k1} - 2b_kw_{k1} & \ldots & w_{kD} - 2b_kw_{kD} & -w_{k1}^2 & -w_{k1}w_{k2} & \ldots & -w_{kD}^2 \end{bmatrix}.$$

In conclusion, h is a linear transformation of $\phi(x)$ because h is obtained by applying the matrix A_Θ, which depends on the model parameters A_Θ,to the feature transformation $\phi(x)$, which is independent of A_Θ.

3.2 - Output $\hat{y}$ as a Linear Transformation of the Feature Mapping $\phi(x)$

3.3 - Equivalence of Parameters c and θ with Precision ε

For this task, we began by considering the activation function $g(z) = z(1 - z)$, which introduced quadratic terms into the network's computation. The hidden layer output h was given by:

$$h = g(Wx + b) = (Wx + b) \odot (1 - Wx - b),$$

where $\odot$ represented element-wise multiplication. By expanding each element h_i, we obtained:

$$h_i = (W_i x + b_i)(1 - W_i x - b_i) = (W_i x + b_i) - (W_i x + b_i)^2.$$

This showed that the hidden layer output included both linear and quadratic terms involving x.

Next, we expressed the predicted output of the neural network as:

$$\hat{y} = v^\top h + v_0 = \sum_{i=1}^{K} v_i h_i + v_0.$$

Substituting h_i into this expression, we expanded $\hat{y}$ as:

$$\hat{y} = \sum_{i=1}^{K} v_i \left[(W_i x + b_i) - (W_i x + b_i)^2\right] + v_0.$$

To simplify further, we expanded the squared term $(W_i x + b_i)^2$ as:

$$(W_i x + b_i)^2 = (W_i x)^2 + 2 b_i W_i x + b_i^2.$$

Thus, the predicted output became:

$$\hat{y} = \sum_{i=1}^{K} v_i \left[W_i x + b_i - (W_i x)^2 - 2 b_i W_i x - b_i^2\right] + v_0.$$

After collecting terms by their degree, we rewrote the expression as:

$$\hat{y} = -\sum_{i=1}^{K} v_i (W_i x)^2 - 2 \sum_{i=1}^{K} v_i b_i W_i x + \sum_{i=1}^{K} v_i W_i x + \sum_{i=1}^{K} v_i b_i - \sum_{i=1}^{K} v_i b_i^2 + v_0.$$

At this point, we grouped similar terms to obtain the following form:

$$\hat{y} = x^\top A x + d^\top x + e,$$

where the components A, d, and e were defined as:

$$A = -\sum_{i=1}^{K} v_i W_i^\top W_i, \quad d = \sum_{i=1}^{K} v_i (W_i - 2 b_i W_i), \quad e = \sum_{i=1}^{K} v_i (b_i - b_i^2) + v_0.$$

The vector $c \in R^{\frac{(D+1)(D+2)}{2}}$ contained all the unique elements of A, d, and e, corresponding to the coefficients of the quadratic, linear, and constant terms.

To determine the parameters $\Theta = (W, b, v, v_0)$, we first selected W such that it spanned the space of quadratic terms. Since $K \geq D$, we ensured that W was full rank. In cases where adjustments were needed, we perturbed W slightly to approximate the desired structure within a precision ϵ.

Using the expression for A, we solved for v, ensuring that the relationship:

$$A \approx -\sum_{i=1}^{K} v_i W_i^\top W_i$$

held true. From the linear coefficients d, we computed b such that:

$$d = \sum_{i=1}^{K} v_i (W_i - 2b_i W_i).$$

Finally, we determined v_0 using the constant term e as follows:

$$e = \sum_{i=1}^{K} v_i (b_i - b_i^2) + v_0.$$

To ensure that the reconstructed parameters matched the vector c with precision ϵ, we adjusted Θ as necessary. Since any matrix A could be perturbed slightly to become non-singular, we verified that:

$$\|c_\Theta - c\| < \epsilon.$$

In summary, we expressed the network's output as a quadratic function of x and derived the parameters $\Theta = (W, b, v, v_0)$ by systematically solving for A, d, and e. This approach demonstrated that the neural network, parameterized by Θ, could approximate any target vector c within arbitrary precision ϵ.

3.4 - Closed-Form Solution for $\hat{c}_\Theta$ and Uniqueness of the Problem

First, we assumed that the model output $\hat{y}(\mathbf{x}; c_\Theta)$ could be expressed as a linear function of the parameter vector c_Θ. Specifically, we represented $\hat{y}(\mathbf{x}; c_\Theta)$ as:

$$\hat{y}(\mathbf{x}; c_\Theta) = \mathbf{f}(\mathbf{x})^\top c_\Theta,$$

where $\mathbf{f}(\mathbf{x})$ is a feature vector containing polynomial terms of $\mathbf{x}$, and c_Θ is the vector of parameters.

Given this linear structure, the loss function simplifies to:

$$L(c_\Theta; \mathcal{D}) = \frac{1}{2} \sum_{n=1}^{N} \left(\mathbf{f}(\mathbf{x}_n)^\top c_\Theta - y_n \right)^2.$$

To minimize the loss, we computed the gradient of L with respect to c_Θ and set it equal to zero. This yielded:

$$\nabla_{c_\Theta} L = \sum_{n=1}^{N} \left(\mathbf{f}(\mathbf{x}_n)^\top c_\Theta - y_n \right) \mathbf{f}(\mathbf{x}_n) = \mathbf{0}.$$

Simplifying this expression, we obtained the normal equations:

$$\left(\sum_{n=1}^{N} \mathbf{f}(\mathbf{x}_n) \mathbf{f}(\mathbf{x}_n)^\top \right) c_\Theta = \sum_{n=1}^{N} y_n \mathbf{f}(\mathbf{x}_n).$$

Next, we formulated the problem in matrix form. Defining:

$$\mathbf{F} = \begin{bmatrix} \mathbf{f}(\mathbf{x}_1)^\top \\ \mathbf{f}(\mathbf{x}_2)^\top \\ \vdots \\ \mathbf{f}(\mathbf{x}_N)^\top \end{bmatrix} \quad \text{and} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix},$$

where $\mathbf{F}$ is an $N \times M$ matrix with $M = \frac{(D+1)(D+2)}{2}$, the normal equations became:

$$\mathbf{F}^\top \mathbf{F} c_\Theta = \mathbf{F}^\top \mathbf{y}.$$

Since $N > M$ and the input vectors $\{\mathbf{x}_n\}$ are linearly independent, the matrix $\mathbf{F}^\top \mathbf{F}$ is symmetric, positive definite, and invertible. Therefore, we solved for c_Θ using the closed-form solution:

$$\hat{c}_\Theta = \left(\mathbf{F}^\top \mathbf{F} \right)^{-1} \mathbf{F}^\top \mathbf{y}.$$

Why Global Minimization is Achievable:

In general, feedforward neural networks pose an intractable problem for global minimization due to the non-convexity of the loss function. The presence of nonlinear activation functions and complex parameter interactions often results in multiple local minima and saddle points, which makes optimization challenging.

However, in this specific case, the problem becomes tractable due to the following reasons:

- The activation function $g(z) = z(1-z)$ allows the network output $\hat{y}$ to be reparameterized as a linear function of c_Θ.
- The loss function $L(c_\Theta; \mathcal{D})$ becomes a quadratic function of the parameters c_Θ. Quadratic loss functions are convex and, therefore, have a unique global minimum.
- The input vectors $\{\mathbf{x}_n\}$ are linearly independent, ensuring that the design matrix $\mathbf{F}^\top\mathbf{F}$ is invertible.

These properties collectively ensure that the minimization problem admits a closed-form solution for $\hat{c}_\Theta$, making the optimization process straightforward and tractable.